Real-Time Hyperspectral Image Compression Onto Embedded GPUs

María Díaz, Raúl Guerra, Pablo Horstrand, Ernestina Martel, Sebastián López, *Senior Member, IEEE*, José F. López, and Roberto Sarmiento

***Abstract*—Real-time hyperspectral imaging on-board compression represents a critical processing step in many remote sensing applications where the acquired hyperspectral data need to be efficiently stored and/or transferred. However, the complexity of the compression algorithms as well as the volume of data to be compressed and the limited computational resources of the hardware devices available on-board turn the real-time compression into a very challenging task. This paper presents a low-power-consumption solution for real-time lossy compression of hyperspectral images. The lossy compression algorithm for hyperspectral image system (HyperLCA) compressor has been implemented onto two NVIDIA Jetson developer kits. These NVIDIA boards include low-power embedded graphic processing units, which allow parallel programing for speeding up the compression process at a reasonable low power consumption. The experiments carried out in this paper are oriented to the necessities imposed by a specific smart farming application although all drawn conclusions are extrapolable to other fields in which remotely sensed hyperspectral images are to be compressed in real time. The obtained results verify both the good performance of the HyperLCA compressor for the targeted application and the achievement of a real-time performance by using the developed implementations. Additionally, several comparisons and conclusions have been drawn from the experiments in relation to the different strategies employed for accelerating the compression process.**

***Index Terms*—CUDA, embedded systems, hyperspectral compression, lossy compression, low-power graphical processing units (LPGPUs), real time, unmanned aerial vehicle (UAV).**

I. Introduction

THE capability of the hyperspectral sensors for collecting information across the electromagnetic spectrum provides very useful information for many remote sensing applications. However, depending on the characteristics of the applications, the huge amount of data collected by these sensors must be stored onboard or directly transferred to the earth surface. Since the bandwidth of the connection is limited, as well as the memory, power, and computational capabilities of the remote sensing hyperspectral acquisition systems, the on-board hyperspectral image compression is a very important task, which presents several challenges.

First, because of the amount of data contained in these images, it is important to achieve high compression ratios (CRs), but without losing too much information. Second, the high data rate provided by the new-generation sensors imposes the necessity of carrying out a real-time compression process in order to efficiently store and/or transfer the acquired data without unnecessarily accumulating high amounts of uncompressed data. Third, the limited computational power makes compulsory the development of low-complexity approaches for hyperspectral image compression. In addition, most of nowadays remote sensing applications employ pushbroom/whiskbroom scanners [1]–[4], which collect the hyperspectral images in a line-by-line fashion. This generates a strong necessity of hyperspectral compression solutions, which permit to compress blocks of pixels without any specific spatial alignment requirement in order to ease a real-time compression performance.

In order to cope with the high data-rate and achieve high CRs, it is necessary to use lossy compression techniques against lossless or near-lossless approaches, which shows very limited CRs of about 2$\sim$3:1 [5]. In addition, transform-based approaches [6], such as discrete wavelet transform [7], [8] or the Karhunen–Loève Transform [9], [10] are generally preferred to spatially/spectrally decorrelate the images for lossy compressor. Unlike this kind of decorrelators, prediction-based techniques [11] use nearby or neighboring samples to predict each sample value, which makes them not very suitable for real-time compression employing pushbroom/whiskbroom scanners since spatial information is required.

Although most of the state-of-the-art lossy compressors achieve very satisfactory performance in terms of rate-distortion, they are characterized by extremely high computational costs, intensive memory requirements, and a nonscalable nature, which prevents their use in applications under latency/power/memory constrained environments, such as on-board compression. For these reasons, the lossy compression algorithm for hyperspectral image systems (HyperLCA) [12] was introduced as a

Manuscript received November 14, 2018; revised April 12, 2019; accepted May 4, 2019. Date of publication June 5, 2019; date of current version September 16, 2019. This work was supported in part by the European Commission through the Electronic Components and Systems for European Leadership Joint Undertaking (ENABLE-S3 project, no. 692455) and in part by the Ministry of Economy and Competitiveness of the Spanish Government (ENABLE-S3 project, no. PCIN-2015-225; and PLATINO project, no. TEC2017-86722-C4-1-R). This work of M. Díaz was supported by the Predoctoral Grant given by the Agencia Canaria de Investigacion, Innovacion y Sociedad de la Informacion of the "Conserjería de Economía, Industria, Comercio y Conocimiento" of the "Gobierno de Canarias," which is supported in part by the European Social Fund (POC 2014-2020, Eje 3 Tema Prioritario 74 (85%)). *(Corresponding author: Sebastian Lopez.)*

The authors are with the Institute for Applied Microelectronics, University of Las Palmas de Gran Canaria, 35017 Las Palmas, Spain (e-mail: mdmartin@iuma.ulpgc.es; rguerra@iuma.ulpgc.es; phorstrand@iuma.ulpgc.es; emartel@iuma.ulpgc.es; seblopez@iuma.ulpgc.es; lopez@iuma.ulpgc.es; roberto@iuma.ulpgc.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSTARS.2019.2917088

low-computational complexity alternative, which provides high CRs with a good compression performance at a reasonable computational burden. Additionally, this transform-based compressor independently processes blocks of hyperspectral pixels without any required spatial alignment between them, which makes it an ideal candidate for the real-time compression of hyperspectral data taken by pushbroom/whiskbroom scanners.

In general terms, we present in this paper a design methodology, which ensures the lossy compression of hyperspectral data for applications in real-time characterized by high data-rates. Likewise, it is focused on remote sensing applications where the available computational resources are limited, due to power, weight, or space limitations. Concretely, we present a smart farming application where a visible-near-infrared (VNIR) hyperspectral pushbroom scanner is mounted onto an unmanned aerial vehicle (UAV) for collecting periodical information of the crops, which results in a huge amount of data that need to be managed, processed, and analyzed. The employed camera is able to collect information from 400 to 1000 nm using 224 spectral bands and 1024 spatial pixels per frame. It provides a maximum frame rate of 330 frames/s (F/S), what results in 144.375 MB/s (more than 8 GB/min). In addition, this UAV carries a processing board, which manages many tasks at the same time, such as, the data acquisition, data calibration, data storing, and/or transferring processes, camera controlling, and also the drone flight control. Some of the limitations of this application are related to the limited power available as well as the limited size and weight that may be efficiently carried by the drone. Because of this reason, we have selected the NVIDIA Jetson TK1 and the Jetson TX2 processing boards [13], [14], which provide a reasonable computational power at a relatively low power consumption. These boards include a low-power graphical processing unit (LPGPU) that allows parallel programming for speeding up the executed processes. This is especially useful for accelerating the compression of the collected hyperspectral data.

In this paper, we have deeply analyzed the different stages of the HyperLCA compressor in terms of floating point operations (FLOPs) and serial execution times in order to identify those stages more suitable for being accelerating in the available LPGPUs. Finally, the HyperLCA Transform, which is the most demanding part of the HyperLCA compressor, has been implemented in the LPGPUs available in the Jetson boards through the use of NVIDIA CUDA programming language. The efficiency of the developed kernels has been measured for the different available configurations of the HyperLCA compressor using the NVIDIA profiling tool. Additionally, since the LPGPUs present in the Jetson TK1 and the Jetson TX2 boards have different architectures, Kepler and Pascal, the behavior of the developed kernels has been compared, in terms of efficiency, speed, and resources used, in both architectures.

Furthermore, three different implementation models of the whole compression model have been studied, seeing them as an evolution toward an optimal configuration, which fulfills the constraints of the targeted application. These strategies are focused on exploiting the parallelism of the HyperLCA compressor beyond the thread level parallelism inherent to the GPU programming model, more concretely, pipelining the execution of the compression stages of the HyperLCA algorithm for the different blocks of pixels as well as the communications and memory transfers. The efficiency of these implementation models has been evaluated measuring the time required for compressing a set of real hyperspectral images collected by the described acquisition system. The obtained results verify that the implementation of the HyperLCA algorithm carried out in this paper is able to produce real-time compression results for the described acquisition system and for the different targeted configurations of the HyperLCA compressor. Additionally, a huge amount of conclusions and comparisons have been drawn from this paper in relation with different possible approaches for exploiting the resources of the Jetson TK1 and the Jetson TX2 NVIDIA boards, considering the efficiency, the speed, and the amount of employed resources.

As far as we know, it is the first time that a low-power consumption solution is given for streaming lossy compression of hyperspectral images based on transform-based compression techniques. Nascimento *et al.* [15]–[17] used embedded LPGPUs to implement a compress sensing method but it does not fulfill all the requirements imposed in this paper. For instance, the employed algorithm needs to know in advance the number of endmembers present in the whole image. In addition, just synthetic images are used in the study and a fair comparison is not possible. Davidson and Bridges [18] investigates the error resilience of a GPU image processing application for space where the standard lossless image compression algorithm consultative committee for space data systems (CCSDS)-123 [19] is implemented in an embedded LPGPU. However, since it is a lossless solution, achieved CRs are very limited. Going far away from low-power consumption solutions, we can find same GPU implementations of lossy compressors in the literature. In [20], the lossless and lossy modes of the JPEG2000 are implemented in a standard desktop's GPU. However, it requires spatial information through the performance of a spatial transform and hence, a real-time compression is not guaranteed. In addition, the achieved CRs are very limited compared with those obtained in this paper. A similar work is done in [21] with a prediction-based lossy compressor. As it can be seen, few state-of-the-art works are done in the field of lossy compression for hyperspectral imagery with GPUs. It is not the same for lossless compression where many publications can be found [22]–[25].

This paper is organized as follows. Section II introduces the characteristics of the HyperLCA compressor that are more relevant to the execution and understanding of this paper. Section III makes a brief introduction to the basic characteristics of the NVIDIA architectures that need to be considered in order to develop an implementation that achieves an efficient use of their resources. Section IV explains in detail the different implementation models carried out in this paper. Sections V and VI display the obtained results and the conclusions.

II. Description of the HyperLCA Algorithm

The HyperLCA algorithm is a lossy transform-based compressor specially designed to achieve high compression rate-distortion ratios at a reasonable low computational burden and a high level of parallelism for hyperspectral remote sensing

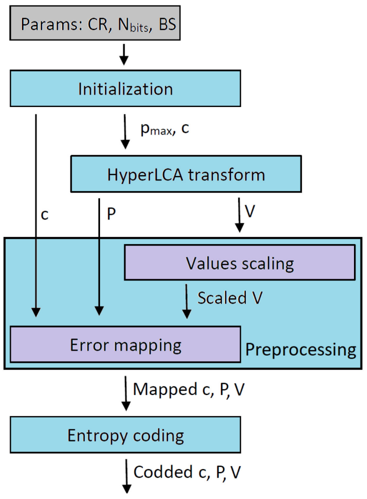


Fig. 1. Diagram of the HyperLCA algorithm compression stages.

applications, which offers many advantages over some other state-of-the-art lossy compressors [12]. The method followed by the HyperLCA compressor is an unmixing like strategy that allows compressing with higher precision the image pixels that are potentially more useful for the ulterior hyperspectral applications. Indeed, this strategy has been proved to be useful for multiple hyperspectral imaging applications [26]–[28]. In addition, the HyperLCA compressor permits to independently compress blocks of image pixels since no spatial information is required.

Fig. 1 shows the different stages involved in the compression process performed by the HyperLCA algorithm: an initialization stage where some parameters are configured, a spectral transform, a preprocessing stage, and an entropy coding stage. The HyperLCA spectral transform sequentially selects the most different pixels of the hyperspectral data set using orthogonal projection techniques. The set of selected pixels is then used for projecting the hyperspectral image, obtaining a spectral decorrelated and compressed version of the data. The HyperLCA preprocessing stage is executed after the HyperLCA Transform for adapting the output data for being entropy coded in a more efficient way. Finally, the HyperLCA entropy coding stage manages the codification of the extracted vectors using a Golomb–Rice coding strategy.

A. HyperLCA Input Parameters

The HyperLCA algorithm needs three main input parameters to be configured.

1) *Minimum desired CR* defined as the relation between the number of bits in the real image and the number of bits of the compressed data.
2) *Block size (BS)*, which indicates the number of hyperspectral pixels that compose each block of pixels that the HyperLCA compressor independently compresses.
3) *Number of bits used for scaling the projection vectors (N_{bits}).* This value determines the precision and dynamic range to be used for representing the values of *V* vectors.

Algorithm 1: HyperLCA Transform.

```
Inputs:
$M = [r_1, \ldots, r_{\text{BS}}]$, $p_{\max}$, $c$
Outputs:
$P = [c, p_1, \ldots, p_{p_{\max}}]$, $V = [v_1, \ldots, v_{p_{\max}}]$
Declarations:
$P = [p_1, \ldots, p_{p_{\max}}]$; {Extracted pixels.}
$V = [v_1, \ldots, v_{p_{\max}}]$; {Projected image vectors.}
$M_c = [x_1, \ldots, x_{\text{BS}}]$ {Centralized version of $M$}
Algorithm:
1:  {Additional stopping condition initialization.}
2:  for $i = 1$ to $p_{\max}$ do
3:      for $j = 1$ to $BS$ do
4:          $b_j = {x_j}^t \cdot x_j$
5:      end for
6:      $j_{\max} = \arg\max(b_j)$
7:      $p_i = r_{j_{\max}}$
8:      $q = x_{j_{\max}}$
9:      $u = x_{j_{\max}} / ((x_{j_{\max}})^t \cdot x_{j_{\max}})$
10:     $v_i = u^t \cdot M_c$
11:     $M_c = M_c - q \cdot v_i$
12:     {Additional stopping condition checking.}
13: end for
```

B. HyperLCA Initialization

Once that the HyperLCA compressor has been correctly configured, it first determines the number of pixels vectors and projection vectors ($p_{\max}$) to be extracted for each block, as shown in (1), where *DR* refers to the number of bits per pixel per band of the hyperspectral image to be compressed and N_b the number of spectral bands of the hyperspectral image. The extracted pixel vectors are referred as *P* and the projection vectors are referred as *V* in the rest of this paper. Once that $p_{\max}$ has been obtained, the average pixel, also called centroid, *c*, is computed for each block of pixels to be compressed. These data are used as inputs of the HyperLCA Transform, which is the most relevant part of the HyperLCA compressor

$$p_{\max} \leq \frac{\text{DR} \cdot N_b \cdot (\text{BS} - \text{CR})}{\text{CR} \cdot (\text{DR} \cdot N_b + N_{\text{bits}} \cdot \text{BS})}. \tag{1}$$

As shown in (1), the compression achieved within this process directly depends on the number of selected pixels, $p_{\max}$. Selecting more pixels provides better decompressed images but lower CRs.

C. HyperLCA Transform

The HyperLCA Transform stage provides most of the CR obtained by the HyperLCA compressor and also, most of its flexibility and advantages. Additionally, it is the only lossy part of the HyperLCA compression process. The HyperLCA Transform is described in detail in Algorithm 1.

First of all, the HyperLCA Transform subtracts the average pixel (c) to all the pixels of the block to be compressed, obtaining the centralized image block (M_c). After doing so, the HyperLCA compression process mainly consists of three steps, which are sequentially repeated. First, the brightest pixel in M_c, which is the pixel with more remaining information, p_i, is selected (lines 2 to 7 of Algorithm 1). After doing so, the vector v_i is calculated as the projection of the image, M_c, in the direction spanned by p_i (lines 8 to 10 of Algorithm 1). Finally, the information of the image that can be represented with the extracted p_i and v_i vectors is subtracted from the M_c, as shown in line 11 of Algorithm 1.

Accordingly, M_c contains the information that is not representable with the already selected pixels, P, and V vectors. Hence, the values of M_c in a particular iteration, i, would be the information lost in the compression–decompression process if no more pixels p_i and v_i vectors are extracted. This fact makes it relatively simple to add extra stopping conditions based on quality metrics such as the maximum absolute difference (MAD), or the signal to noise ratio (SNR). These metrics would check in every iteration of the HyperLCA Transform if the amount of information remaining in M_c is high, and more p_i and v_i vectors are required, or if it is low enough and the algorithm may stop.

In this paper, no extra stopping conditions based on quality metrics have been added, and hence, $p_{\max}$ iterations are executed for every block of pixels, extracting the same number of P and V vectors. The goal is to guarantee a relatively constant compressed data rate at the lowest possible computational complexity in order to satisfy the requirements imposed by the targeted application.

D. HyperLCA Preprocessing

This stage is crucial in the HyperLCA compression algorithm since the HyperLCA Transform output data are adapted for being entropy coded in a more efficient way. This compression stage encompasses two different parts.

1) Scaling V Vectors: Unlike the average pixel, c, and the real hyperspectral pixels, P, elements contained in the V vectors are not integers. As further explained in [12], the V vectors contain the projection of the image pixels into the space spanned by the different orthogonal projection vector u in each iteration. This results in values of V vector elements between -1 and 1. Hence, in order to fully exploit the dynamic range available according to the N_{bits} used for representing these vectors and to avoid losing too much precision in the conversion, vectors V can be easily scaled, as shown below

$$v_{j_{\text{scaled}}} = (v_j + 1) \cdot (2^{N_{\text{bits}}-1} - 1). \tag{2}$$

After doing so, the scaled vectors V are rounded to the closest integer values.

2) Error Mapping: The entropy coding stage takes advantage of the redundancies within the data to assign the shortest word length to the most common values in order to achieve higher CRs. In order to facilitate the effectiveness of this stage, the output vectors of the HyperLCA Transform, after the preprocessing of V vectors, are independently lossless processed in order to represent their values using only positive integer values closer to zero than the original ones, using the same dynamic range. To do this, the HyperLCA algorithm makes use of the prediction error mapper described in the CCSDS recommended standard for lossless multispectral and hyperspectral image compression [29].

Header
Block 1
Block 2
Block 3
. . .

Fig. 2. General structure of the bitstream generated by the HyperLCA algorithm.

E. HyperLCA Entropy Coding

The last stage of the HyperLCA compressor corresponds to a lossless entropy coding strategy. The HyperLCA algorithm makes use of the Colomb–Rice algorithm [30] where each single output vector is independently coded. To do this, first the compression parameter, M, is calculated as the average value of the vector. Second, the lowest power of 2 higher than M is calculated as $b = \log_2(M) + 1$. Then, each value is divided by M and the division quotient, q, and the reminder, r, are obtained. The quotient, q, is later coded using a unary code, whereas the remainder, r, is coded as a plain binary using b bits if r is greater or equal to $2^b - M$, or $b - 1$ bits if it is not.

F. HyperLCA Bitstream Generation

Finally, the outputs of the aforementioned compression stages are packaged in the order that they are produced, generating the compressed bitstream whose structure is graphically shown in Fig. 2.

The header contains the global information about the hyperspectral image and the parameters used in the HyperLCA compression process, such as: number of columns, rows and bands of the hyperspectral image (N_c, N_r, and N_b), BS, $p_{\max}$, DR, N_{bits}, and one extra bit, which indicates if any additional stopping condition is used. The individual blocks may be packaged in two different ways, depending if any extra stopping condition, based on quality metrics, is used or not. When no extra stopping condition is used, the number of P and V vectors contained in each block package is the same, $p_{\max}$, and its value can be sent just once in the bitstream header. Otherwise, if any extra stopping condition is used, the number of P and V vectors contained in each block package, $p \leq p_{\max}$, may be different for each block and need to be specified in each block package.

III. Graphics Processing Units

A GPU can be understood as an array of independent processors, which correspond to an independent execution thread. Each of these execution threads can only execute one operation per cycle. However, the high parallelism inherent to GPUs arises from their capability of executing the same operation in many different threads at the same time using different data.

In this paper, the HyperLCA lossy compressor has been implemented onto two NVIDIA LPGPUs using the CUDA programming model for NVIDIA GPUs. In order to better understand the GPU architectures and their parallelism, the basic components of any NVIDIA GPU architecture will be briefly described in Sections III-A and III-B.

A. GPU Architecture

A NVIDIA GPU is built around a scalable array of *streaming multiprocessors* (SMs), which support the concurrent execution of multiple threads. The replication of this architectural block leads to the high hardware parallelism of the GPUs [31]. The basic components, which compose an SM, are: execution units or CUDA cores, shared memory/L1 cache, register file, load/store units, special function units, instruction dispatch units, and warp schedulers.

From a software point of view, GPU threads are organized into thread blocks when a kernel grid is launched in the CUDA programming model. These CUDA thread blocks are a pure software concept, which actually do not exist in the hardware design. From a hardware point of view, these CUDA thread blocks are scheduled on only one SM and an SM can handle more than one thread block at the same time. Once a thread block is assigned to an SM, CUDA manages these threads in groups of 32, called warps. All threads in a warp execute the same instructions at the same time. The warp schedulers select active warps on each clock cycle and dispatch them to execution units. After several clock cycles, a pipeline of instructions is scheduling within the SM. The objective of this pipeline is to hide the instruction latencies. Hence, it is mandatory that some warps are available in every clock-cycle to launch instructions in them while other warps are busy running previous instructions. However, this ideal situation may be limited by different GPU resources, causing gaps of time where no eligible warps are available in a clock cycle and consequently, new instructions cannot be launched. Some factors, which negatively affect the instructions pipeline, are as follows.

1) The number of resident warps per SM, as well as, the number of resident thread blocks and threads. Generally, it is desired to achieve a high number of active warps per SM in such a way that it is more likely to find eligible warps in every clock cycle.
2) Shared memory and register files. The amount of shared memory is fixed and partitioned among the thread blocks scheduled on an SM while register file is partitioned among the threads. Hence, the higher the consumption of registers and shared memory by threads and threads blocks, the fewer warps that can be simultaneously scheduled on an SM.

B. Streams and Concurrency

CUDA programming model permits the concurrent execution of kernels and memory transfers through a mechanism named *CUDA Streams*. A CUDA stream refers to a queue of operations, such as kernel launches, host-device data transfers, and other commands, which are executed in the device in a strict ordering managed by the host code. However, the execution of operations in a stream is totally asynchronous with respect to the host and operations running in other streams. It means that each stream may execute its operations out of order with respect to other streams. As a consequence, it permits overlapping multiple kernel launches and data transfers, which are being executed in different streams, if there are enough GPU resources available.

However, if no stream is specified, all kernel launches and data transfers are implicitly queued in a default stream named *NULL stream*. The use of this default stream implies that all kernel launches and memory transfers are blocking calls and therefore, serially executed in the Host-Device model. It means that the aforementioned concurrency of multiple GPU operations is lost. As a consequence, those operations managed by nondefault streams are blocked until the default stream is idle. Therefore, if the pipelining of multiple GPU operations is desired, nondefault streams must be explicitly created and the GPU operations to be performed must be assigned to them.

The use of nondefault streams is a powerful tool of the CUDA programming model, which allows to deepen in the parallelism offered by the GPUs. Hence, it will be further exploited along this paper in order to ensure that the assigned goals are achieved.

IV. Methodology

In this section, the different methodologies followed to implement the whole HyperLCA lossy compressor are extensively described. First, the computational complexity of the HyperLCA compressor is evaluated in terms of the number of FLOPs involved in each stage of the algorithm. Second, a GPU implementation of the HyperLCA Transform stage has been adequately handled in Section IV-B following the traditional Host-Device CUDA programming model. Third, different implementation models of the whole HyperLCA compressor have been addressed in Section IV-D, using additional parallelization strategies beyond the thread level parallelism inherent to the GPU programming model. These parallelization strategies aim to make a more efficient use of the resources available in the Jetson TK1 and Jetson TX2 boards.

It must be mentioned that the different stages of the HyperLCA compressor have been restructured along this section. The four main stages of the HyperLCA compressor, described in Section II and shown in Fig. 1, were defined from an algorithmic point of view. However, when the operations involved in these parts of the HyperLCA algorithm are implemented using a Host–Device model, they can be grouped depending of whether they are potential candidates for being executed in the host domain [central processing unit (CPU)], or they are more prone to be executed in the device (LPGPU) because of their higher parallelizable nature. Hence, from an implementation and execution point of view, the HyperLCA algorithm has been structured in three main stages.

1) *Initialization* where $p_{\max}$ is calculated following (1). This operation consists of just a simple division where a configuration setting is defined and besides, it is computed once at the beginning of the compression process with independence of the number of blocks to be compressed.
2) *HyperLCA Transform*, which involves not only the operations of the HyperLCA Transform described in Section II, but also the computation of the average pixel and the scaling of V vector. These operations are the most computational demanding parts of the algorithm but, in return, they are highly parallelizable to be accelerated in the LPGPU, as it is further explained in Sections IV-A and IV-B.
3) *Coder*, which involves the HyperLCA entropy coding stage and the error mapping step. This process is much

TABLE I
NUMBER OF FLOPs REQUIRED BY THE HYPERLCA TRANSFORM AND THE CODER STAGES TO PROCESS A SINGLE BLOCK OF BS HYPERSPECTRAL PIXELS

Stage	FLOPs
HyperLCA Transform	$p_{\max} \cdot (6 \cdot N_b \cdot BS + N_b) + 2 \cdot N_b \cdot BS + N_b$
Coder	$5 \cdot (p_{\max}+1) \cdot N_b + 5 \cdot p_{\max} \cdot BS + 17 \cdot (2 \cdot p_{\max}+1)$

less computational demanding than the HyperLCA Transform and it has a more sequential nature, as it is further explained in Section IV-A.

A. Computational Complexity of the HyperLCA Compressor

In this section, the computational complexity of the HyperLCA compressor is evaluated in terms of the number of FLOPs involved in each stage of the algorithm. The goal of this study is to identify which parts of the algorithm are more computationally intensive and hence, more suitable for being accelerated and executed in the LPGPU in order to increase the overall speed-up of the compression process.

Table I summarizes the number of FLOPs required by the HyperLCA Transform and the Coder stages to process a single block of BS hyperspectral pixels. Since the number of FLOPs required by the Initialization stage is negligible, it has not been included in Table I. Considering that $p_{\max}$ is considerably much smaller than BS and N_b, it is easy to see that the computation of the HyperLCA Transform involves much more FLOPs than the Coder.

In order to facilitate the comparison, we have added some numeric examples in Table II using the test bench described in Section V-A and different configurations of the input parameters of the HyperLCA compressor. In this case, we have counted the average number of FLOPs required to perform the HyperLCA Transform and the Coder stages for a complete hyperspectral frame. In addition, we have also counted the average percentage of the total serial time required to compute the entire compression process taken by the HyperLCA Transform running in both Jetson TK1 and Jetson TX2 boards. Since the sensor employed to collect the test bench captures 1024 spatial pixels per frame, results corresponding to BS $=$ 256 and BS $=$ 512 are the addition of respectively processing 4 and 2 hyperspectral image blocks with BS pixels each.

From results collected in Table II, we can see that the number of FLOPs required by the HyperLCA Transform is up to three orders of magnitude higher than the Coder. In terms of time, the HyperLCA Transform takes more than 95% of the total serial time required to compress a hyperspectral frame, that is, to apply the transform and codify the outputs. Therefore, we can conclude that the HyperLCA Transform is the stage of the HyperLCA compressor, which comprises the greatest number of operations and consequently, it is the most computationally intensive part. Taking advantage of the high level of parallelism of the involved operations, the HyperLCA Transform is the HyperLCA algorithm stage more suitable for being implemented in the LPGPUs embedded in the Jetson boards in order to decrease the overall time required by the compression process and ensures the high frame rates imposed by the targeted application.

TABLE II
EVALUATION OF THE NUMBER OF FLOPs REQUIRED BY THE HYPERLCA TRANSFORM AND THE CODER TO PROCESS A COMPLETE FRAME OF 1024 PIXELS

Inputs HyperLCA			FLOPs		HyperLCA Transform Time (%)	
Nbits	BS	CR	HyperLCA Transform	Coder	Jetson TK1	Jetson TX2
12	1024	12	1.3642E+07	7.3565E+04	95.6327	96.1046
		16	1.0324E+07	5.5403E+04	95.8141	96.3075
		20	8.1115E+06	4.3295E+04	95.9934	96.5582
	512	12	1.1432E+07	7.1714E+04	95.0928	95.4623
		16	9.2192E+06	5.7738E+04	95.2413	95.6356
		20	7.0067E+06	4.3762E+04	95.4674	95.8873
	256	12	9.2225E+06	7.4516E+04	94.0799	94.5964
		16	7.0092E+06	5.6804E+04	94.3009	94.7548
		20	4.7959E+06	3.9092E+04	94.6801	95.1514
8	1024	12	1.9173E+07	1.0384E+05	96.5423	97.0041
		16	1.4748E+07	7.9619E+04	96.6608	97.0824
		20	1.1430E+07	6.1457E+04	96.7815	97.2375
	512	12	1.5857E+07	9.9666E+04	95.9765	96.4439
		16	1.1432E+07	7.1714E+04	96.1416	96.5418
		20	9.2192E+06	5.7738E+04	96.257	96.6703
	256	12	1.1436E+07	9.2228E+04	94.9555	95.4747
		16	8.1158E+06	2.9820E+04	95.1532	95.6440
		20	7.0092E+06	7.7284E+04	95.2566	95.8013

Proportion of the total serial time (%) taken by the HyperLCA Transform to process a complete frame of 1024 pixels.

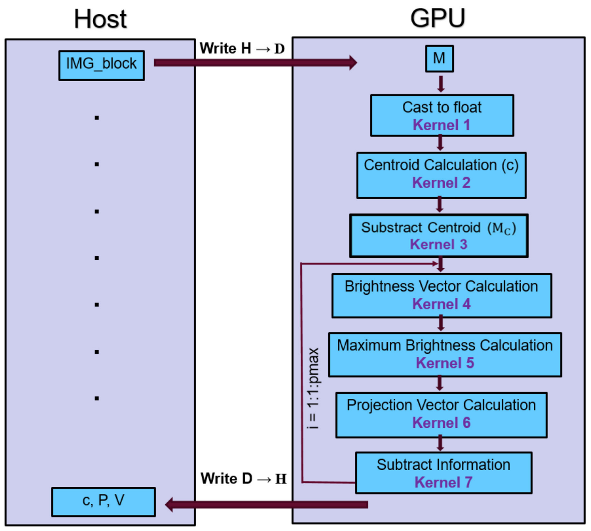


Fig. 3. Flowchart of the HyperLCA Transform in the host–device model.

B. GPU Implementation of the HyperLCA Transform

As it was further explained in Section IV-A, the HyperLCA Transform is the HyperLCA stage more suitable for being parallelized and executed in a GPU. Thus, it has been implemented in the two targeted LPGPUs using CUDA as a design language. The entire HyperLCA Transform has been computed using the host–device CUDA model shown in Fig. 3. This model permits memory transfers that are used to copy the data from the host domain, that is the CPU, to the global memory of the device (the LPGPU for this paper).

The CUDA programming model has been traditionally used for developing applications for peripheral component interconnect express (PCIe) GPUs. In these systems, the GPU memory is separated from the host memory. Accordingly, the traditional host–device model implies the data transfer from the host to the device. In the more recent embedded computing devices, such as Jetson TK1 and Jetson TX2, the physical memory is shared between the GPU and the CPU. In this situation, it is not necessary to copy the data from the host memory to the device memory. Additionally, the latest versions of NVIDIA CUDA offer new mechanisms that allow taking advantage of these situations, such us the use of *unified memory*. However, in the application targeted in this paper, there are many processes running in the Jetson boards that need to access to the same memory positions. For instance, the processes involved in the synchronization and data acquisition need to be able to store the data in memory places that need to be read by the processes involved in processing it, such as the calibration or the compression processes. For doing so, the Linux shared memory has been used.

Because of this reason, for being able to integrate the compression process with the rest of processes of the targeted application, the captured frames are copied from the Linux shared memory spaces to those memory spaces initialized in the style and manner proper to the CUDA programming language, which makes the data understandable and accessible by the processes running in the LPGPUs. Despite the evident disadvantage of introducing one extra copy for each captured frame, this also provides the advantage of following the traditional Host–Device model, copying the data from the host memory to the device memory (which is just another location on the same physical memory in this situation). Accordingly, the implementation of the HyperLCA compressor developed in this paper could be directly compiled for any system that uses a PCIe GPU with its own separated memory.

In general terms, the implementation of the HyperLCA Transform using the Host–Device CUDA model comprises three main stages.

1) *Write $H \to D$:* A preliminary transfer of BS pixels from the input image from the host ($\text{IMG}_{\text{block}}$) to the device ($M$).
2) *Kernels:* Seven different operation blocks or kernels are launched in the GPU to perform the operations involved in the HyperLCA Transform.
3) *Write $D \to H$:* Transfer of the HyperLCA Transform outputs from the device to the host.

The operation blocks or kernels, which encompass the second stage, are briefly outlined below. In addition, the kernel configurations with respect to CUDA thread blocks and CUDA threads are also shown in Table III, where $\lceil \rceil$ means rounding-up to the nearest whole number. Since BS directly affects the fixed number of threads per CUDA block, this parameter has been defined in terms of this HyperLCA compressor input parameter. It is also noted that the employed kernel configurations have been selected taking into account that the maximum number of threads per CUDA block for NVIDIA GPUs with compute capabilities higher than 2.0 is 1024 [32]. In addition, kernels have been

TABLE III
KERNELS CONFIGURATION WITH RESPECT TO CUDA THREADS AND THREAD BLOCKS

Kernel	CUDA thread blocks	CUDA threads
Cast to Float	$\lceil \frac{BS \cdot N_b}{1024} \rceil$	1024
Centroid Calculation	N_b	$2^{(\lceil log_2(BS/2) \rceil)}$
Subtract Centroid	BS	N_b
Brightness Vector Calc.	$\lceil \frac{BS}{\frac{1024}{256/2}} \rceil$	$(\frac{1024}{256/2}, \frac{256}{2})$
Max. Brightness Calc.	1	$2^{(\lceil log_2(BS) \rceil)} \geq 256$
Projection Vector Calc.	$\lceil \frac{BS}{\frac{1024}{256/2}} \rceil$	$(\frac{1024}{256/2}, \frac{256}{2})$
Subtract Information	N_b	BS

defined for hyperspectral images up to 256 spectral bands and for a maximum BS of 1024 pixels.

1) Kernel 1: Cast to float—The HyperLCA Transform operations described in [12] employ floating point arithmetic. Accordingly, the data captured by the hyperspectral sensor, which are stored as unsigned integer values, need to be converted to the floating point. This process is fully parallelizable since each GPU thread independently processes one element of the image block, M, being M the input hyperspectral data of Algorithm 1.
2) Kernel 2: Centroid calculation—This kernel computes the average pixel c of the image block M. To issue this problem, N_b CUDA thread blocks are defined. Threads of a common block independently work with the pixel values within the same spectral band. To compute the mean value per band, a reduction strategy is employed, making use of the shared memory. For this reason, the number of threads has been defined as the power of two whole numbers closer to the half of BS. In addition, the average pixel, c, is rounded to the closest integer value before starting the HyperLCA transform stage. This not only eases the posterior preprocessing and entropy coding stages, since these two stages need to work with integer values, but also guarantees to use the exact same vector when the inverse process is performed to decompress the image, which increases the overall accuracy of the compression–decompression process. Since the HyperLCA Transform uses floating point arithmetic, the result centroid pixel after the rounding is again casted to float.
3) Kernel 3: Subtract centroid—To centralize M, the average pixel c is subtracted to each pixel of image M, getting image M_c, as shown in Algorithm 1. To issue this operation, BS thread blocks of N_b threads are defined where each block independently processes each pixel. Each thread within the same block works with a pixel band, subtracting the information of the corresponding band of vector c.
4) Kernel 4: Brightness vector calculation—This kernel performs operations covered in lines 1–5 of Algorithm 1. As it can be seen, the brightness calculation of a pixel involves the addition of the squares of each of the pixel

bands. To issue this set of operations in just one kernel, two-dimensional thread blocks are defined where x dimension matches up with each image pixel and y dimension with the spectral bands. Once each thread performs the square of each image element, a reduction strategy along y dimension is followed to perform the addition of the squares for each image pixel. To do this, the number of threads along y dimension has been defined as the half of the maximum number of spectral bands, that is, 256/2. A portion of the shared memory is also employed.

5) Kernel 5: Maximum brightness calculation—This kernel calculates the brightest pixel index and its value as shows line 6 of Algorithm 1. In addition, vectors q and u, defined, respectively, in lines 8 and 9, are also computed. To address all these issues, just one CUDA thread block is required. In order to find the brightness pixel index and its value, a reduction strategy is followed making use of the shared memory. Unlike kernel 2, in this case, the number of threads must be defined as the power of two whole numbers closer to BS since we need to work with thread indexes. The q vector corresponds to the M_c pixel with the highest brightness. The u vector is equal to q vector but whose elements have been divided by the maximum brightness. Since kernels have been defined for a maximum of 256 spectral bands and hence, u and q vectors will have 256 elements as maximum, a minimum of 256 CUDA threads is required in this case.
6) Kernel 6: Projection vector calculation—This kernel computes the output V vectors as it is described in line 10 of Algorithm 1. As it can be seen, the problem was tackled in the same way as kernel 4, where the brightness vector calculation is performed. In this case, instead of computing the addition of the squares of each of the pixel bands, each pixel band is multiplied by its homologous in vector u.
7) Kernel 7: Subtract information—This kernel performs operations shown in line 11 of Algorithm 1 where the spectral information of each pixel that can be represented by vector q is subtracted from M_c. Due to this reason, M_c contains the remaining spectral information, which cannot be represented by the previous extracted pixels. To address these operations, N_b blocks of BS threads are defined. Each block independently processes each pixel and each thread within the same block works with a pixel band. Taking advantage of this thread distribution, the scaling of the V vector described in Section II-D1 is also performed following (2).

Since $p_{\max}$ pixels must be extracted, kernels 4 to 7 are launched $p_{\max}$ times for each image block.

C. Some Notes About the GPU Implementation of the HyperLCA Transform

In this section, we would like to clarify a few details about the decisions taken during the kernel definition. It is also important to highlight that this paper is oriented to pushbroom/whiskbroom hyperspectral scanners, which collect the hyperspectral image in a line-by-line fashion. Therefore, the goal of this implementation is to avoid the necessity of storing a high number of hyperspectral pixels until being able to process them.

Additionally, the employed Jetson TK1 and Jetson TX2 boards share the same physical RAM memory between the CPU and the GPU. Accordingly, the maximum amount of data that can be hold in RAM memory for the compression process is limited, especially considering that at the same time, the boards are managing other processes such as the hyperspectral image capturing process and the compressed image storing and/or transferring processes. All these processes stream the data using ring buffers in an asynchronous manner.

For instance, when the capturing process finishes sensing the first frame, this frame is stored in the first position of the capturing ring buffer, and starts capturing and storing the second frame in the second position of this buffer. At this time, the compression process starts processing the data that were stored in the first position of this buffer. Similarly, the results provided by the HyperLCA Transform are stored in another ring buffer from which the codification process reads the data to be coded. Finally, the coded data are stored in another ring buffer from which the storing and/or transferring processes read the information to be stored and/or transferred. In order to avoid losing portions of data if some of these processes momentarily stalls, these buffers need to be able to hold a relatively high amount of data in relation to the amount of data that is processed in one iteration. Hence, if the BS is 1024 pixels with 256 bands each, it means that the ring buffer used in the capturing process has to be N times that size, where N is the number of frames that this buffer can hold. This is also applied to the rest of ring buffers used in this process.

Due to all these reasons, we decided to keep the BS as small as possible without compromising the quality of the compression results and also, the maximum number of spectral bands. Concretely, we have limited the BS to 1024 pixels and the number of spectral bands to 256, as maximum. However, since BS is a parameter inherent to the HyperLCA compressor and does not have to match the pixel-wide swath of the sensor, it is applicable to any pushbroom/whiskbroom hyperspectral scanner, which provides images with less than 256 hyperspectral bands. For instance, this solution could perfectly work with well-known sensors used in remote sensing, such as AVIRIS (whiskbroom scanner, 677 pixel-wide swath, and 224 bands), HYDICE (pushbroom scanner, 320 pixel-wide swath, and 210 bands), CASI (pushbroom scanner, 512 pixel-wide swath, and 288 bands), and HYPERION (pushbrrom scanner, 256 pixel-wide swath, and 242 bands). In addition, it could also perfectly work with most of the hyperspectral sensors collected by Adão *et al.* [2] in its Table II for being coupled with UAVs and certainly with the targeted application in this paper. It is worth to mention that kernels from 1 to 6, except for kernel 5, can be actually launched for a BS up to 2048 pixels. For BS $\geq$ 1024, kernels 5 and 7 must be redefined since the maximum number of CUDA threads per block that the CUDA compiler can managed is 1024 threads.

Finally, we would like to mention that for the experiments made in next sections, we have fixed the BS values to 256, 512, and 1024. Basically, we have selected these configurations since they are multiple of 32, that is, a warp size. As it has been explained in Section III-A, from a hardware point of view, the CUDA thread blocks are scheduled on only one SM of the GPU, and CUDA manages these threads in groups of 32, called warps, where all threads in the same warp execute the same instructions at the same time. In order to define efficient kernels and making the best use of the GPU resources, we have defined CUDA threads blocks whose number of threads are multiple of 32 and, hence, it must be also the case for the BS since it directly affects them, as it can be inferred from Table III.

D. Host–Device Model of the HyperLCA Lossy Compressor

In general terms, the goal of this paper is to design a model, which permits to compress hyperspectral frames in real-time for applications with limitations on computational resources. It means being able to compress each hyperspectral frame in less time that is required for capturing it. In particular, we present an application where a hyperspectral sensor is mounted on a UAV, which captures frames with a very high frame rate, using a computer platform with an embedded LPGPU. Many processes are running at the same time in the computer platform apart from the compression process. On this basis, the main objective of this paper is to accelerate the most demanding parts of the HyperLCA compressor in order to guarantee the high frame rates imposed by the targeted application and at the same time, lessen the computational burden of the CPU, which is executing many other processes apart from the compression one. As a consequence, different configurations of the Host–Device model of the HyperLCA lossy compressor have been studied, seeing them as an evolution toward an optimal configuration, which fulfills the constraints of the targeted application.

The first and simplest approach is to accelerate the second stage of the HyperLCA compressor, corresponding to the HyperLCA Transform, executing it in the LPGPU and giving rise to the host–device model already described in Section IV-B and which is shown in Fig. 3. In this case, we have a process where the three different HyperLCA compressor stages are managed sequentially in the same CPU process but operations involved in the HyperLCA Transform stage are executed in the LPGPU. Here, each block of BS pixels is also serially compressed, one after the other, as Fig. 4 shows. Throughout this document, this configuration is referred as Parallel Model 1.

A more in-depth analysis of the behavior of the different stages of the HyperLCA compressor reveals that the execution of the Coder stage can be performed with total independence of the HyperLCA Transform stage. Once the coder receives the HyperLCA Transform outputs of a pixel block, it can work with them in the background while the HyperLCA Transform stage processes a new block of pixels. Two independent CPU processes have been used to implement this solution. One of them is in charge of executing the HyperLCA Transform, which involves transferring data from the host to the device, launching the kernels that are executed in the LPGPU and transferring back the outputs of the HyperLCA Transform to the CPU memory. The other CPU process is constantly waiting for a new available HyperLCA Transform outputs to perform the coding

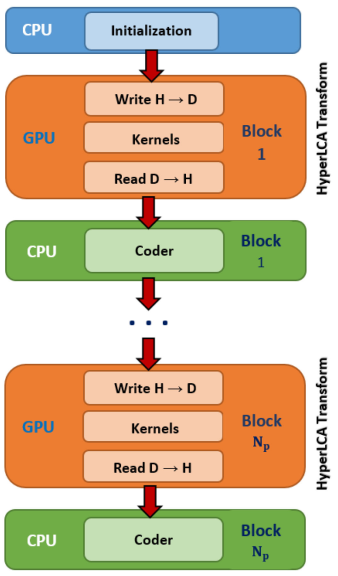


Fig. 4. Parallel Model 1: all HyperLCA compressor stages are sequentially performed in an unique CPU process but the HyperLCA Transform stage is accelerated in the GPU.

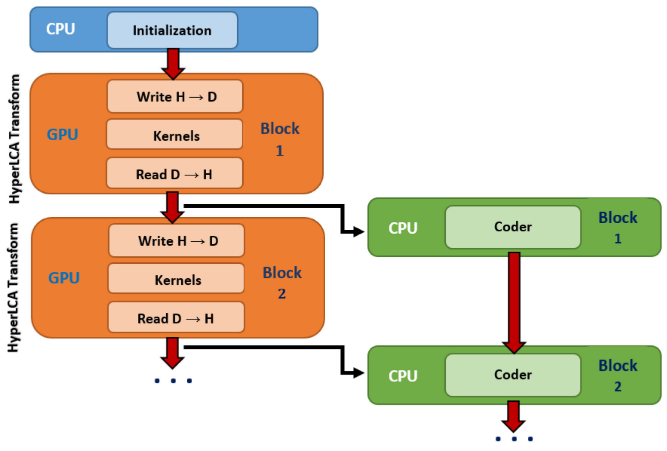


Fig. 5. Parallel Model 2: the HyperLCA Transform and the coding stage are running in two independent CPU processes.

stage of these results. As it was explained in more detail in Section IV-A, the codification is much less computational demanding than the HyperLCA Transform and therefore, it is efficiently executed in the CPU. Throughout this document, this configuration is referred as Parallel Model 2 and its flowchart is shown in Fig. 5.

Finally, the task parallelism introduced by the CUDA streams has been also considered giving rise to the third and the last tested configuration named Parallel Model 3. As it was already introduced in Section III-B, a CUDA stream represents a queue of operations executed in the GPU in a strict and specific order. However, different nondefault streams run asynchronously and concurrently if there are enough available GPU resources. This has been used for pipelining the data transfers between the host and the device and the kernel executions for different block of pixels as shown in Fig. 6. Specifically, one stream, named *GPU*

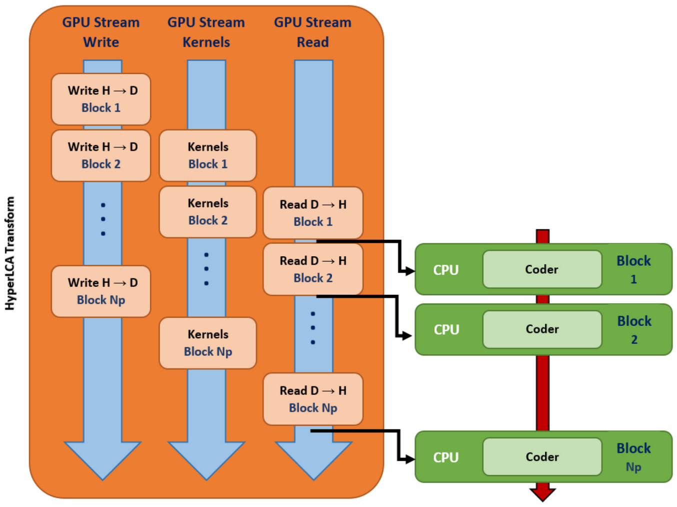


Fig. 6. Parallel Model 3: the HyperLCA Transform has been implemented in the GPU using three nondefault streams while the coding stage is running in another CPU process.

Stream Write, is used for copying the pixel blocks, one by one, from the host to the device. Another stream, named *GPU Stream Kernels*, manages the execution of the different kernels that perform the HyperLCA Transform operations for each pixel blocks. Finally, a third stream, named *GPU Stream Read*, is used for copying the HyperLCA Transform outputs for each pixel block from the device to the host. For instance, according to this parallel model, the copy of the pixel block number 3 from the host to the device, the execution of HyperLCA Transform kernels for the pixel block number 2, and the copy from the device to the host of the HyperLCA Transform results for the pixel block number 1 are concurrently launched. As in Parallel Model 2, the execution of the codification stage is also pipelined with the HyperLCA Transform stage using another CPU process.

In the next sections, these three models will be compared with a serial model where the whole process of the HyperLCA compressor is executed in a unique process of the CPU. Therefore, the three stages of the HyperLCA compressor are managed sequentially, one after the other, giving rise to a host model where the performance of the device is discarded. It means that once the compression of a pixel block has been completed, the next pixel block is processed. Throughout this document, this model is referred as a reference model.

V. Experiments and Results

Several experiments have been carried out in this paper in order to evaluate the performance of the developed implementations of the HyperLCA compressor described in Section IV. First, we have analyzed the suitability of the HyperLCA compressor for performing the lossy compression of hyperspectral data collected by a UAV. Second, the efficiency of the developed kernels have been evaluated using some assessment metrics offered by the NVIDIA profiling tool. Finally, we have assessed the speed-up achieved by the different parallel models described in Section IV-D.

A. Description of the Data Sets

In particular, this paper focuses on processing hyperspectral imaging data collected by a drone for smart farming applications. More specifically, the goal of this paper is to compress the acquired hyperspectral data in real time, in such a way that it can be efficiently transferred to the ground station for its analysis. The high data rate provided by the employed hyperspectral sensor, together with the characteristics of the application and the limited available computational resources, turns the real-time hyperspectral data compression into a very challenging task.

Two real hyperspectral images, collected by a *Specim FX10* [33] hyperspectral camera, have been used for such purpose. This camera is a VNIR pushbroom hyperspectral sensor able to collect information from 400 to 1000 nm using 224 spectral bands and 1024 spatial pixels per frame. The maximum frame rate provided by the Specim FX10 camera is 330 F/S, what results in 144.375 MB/s (more than 8 GB/min), when using Camera Link interface. When using the GigE Vision interface, the maximum speed decreases to 125 MB/s, what still results in more than 7 GB/min.

In this paper, the first 10 bands and the last 34 bands are not being captured because of the low spectral response of the hyperspectral sensor at those wavelengths, what results in just 180 spectral bands being captured. The two data sets contain more than 3000 hyperspectral frames of 1024 pixels and 180 bands each, what results in more than 1 GB/image. The first image, labeled as *Image 1*, was collected at 150 F/S and flying at 45 m of altitude with respect to the takeoff point. The second one, labeled as *Image 2*, was collected at 200 F/S and flying at the same altitude. These images were collected over two different vineyards in Gran Canaria (Canary Islands, Spain).

In order to simplify the experiments, these images have been cropped, extracting two portions of 1024 frames/image. These portions have been labeled as *Image 1—Portion A*, *Image 1—Portion B*, *Image 2—Portion A*, and *Image 2—Portion B*. Fig. 7 shows an RGB representation of these portions, using the bands 113, 45, and 20 of the hyperspectral sensor as the red, green, and blue bands, what approximately corresponds to 700, 540, and 480 nm, respectively.

B. Processing Board

Some of the characteristics needed by the processing board in order to be useful for the targeted smart farming applications can be determined considering just the necessities and limitations imposed by the drone and the hyperspectral imaging acquisition system. In particular, the selected board has to be relatively small and light and it has to demand a relatively low power consumption. Additionally, it has to be able to manage and process in real time the high amount of data provided by the hyperspectral sensor. Due to these reasons, we have put our attention in the NVIDIA Jetson developer boards, and more specifically in the NVIDIA Jetson TK1 [13] and NVIDIA Jetson TX2 [14] developer kits. Despite the power consumed by these devices is low enough for this application [34], [35], these boards include an LPGPU that allows parallel programming for speeding up the executed processes. This is especially useful for accelerating

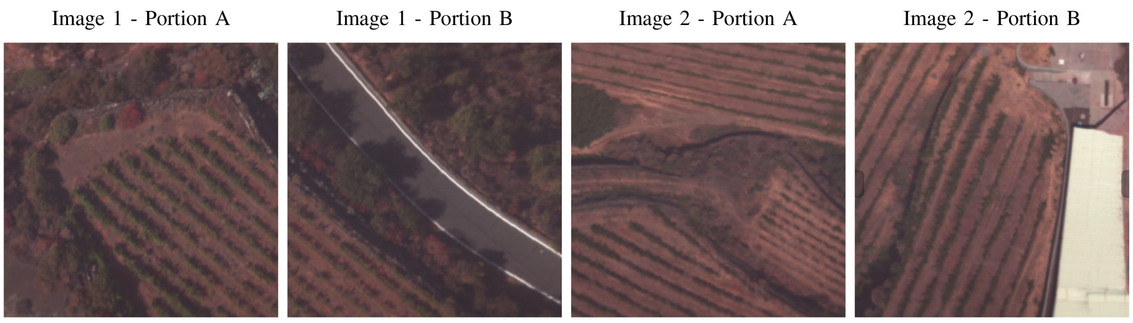


Fig. 7. RGB representation of the hyperspectral images used in the experiments.

TABLE IV
MOST RELEVANT CHARACTERISTICS OF THE NVIDIA JETSON TK1 AND JETSON TX2

	Jetson TK1
LPGPU	GPU NVIDIA Kepler with 192 CUDA cores
CPU	CPU ARM Cortex A15 quad-core NVIDIA 4-Plus-1
Memory	2 GB x16 memory (64 bits bandwidth, 14.93 GB/s) + 16 GB eMMC 4.51 memory
	Jetson TX2
LPGPU	GPU NVIDIA Pascal with 256 CUDA cores
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM A57/2 MB L2
Memory	8 GB LPDDR4 (128 bits bandwidth, 59.7 GB/s)

TABLE V
TECHNICAL SPECIFICATIONS OF THE LPGPUS IN NVIDIA JETSON TK1 AND JETSON TX2

Technical Specifications	Jetson TK1	Jetston TX2
GPU	GK20A	GP10B
Number of CUDA cores	192	256
Numbe of SMs	1	2
Compute Capability	3.2	6.2
Warp Size	32	32
Maximum blocks per SM	16	32
Maximum warps per SM	64	64
Maximum threads per SM	2048	2048
32 bit registers per SM	64K	64K
Maximum shared memory per SM	48KB	64KB

the compression of the collected hyperspectral data. Because of the importance of the characteristics of these boards for the research work covered in this paper, the most relevant details of the NVIDIA Jetson TK1 and Jetson TX2 boards are described in Table IV, whereas Table V collects the most relevant technical specifications of the embedded LPGPUs.

C. Evaluation of the Efficiency of the Developed Kernels

As it was already described in Section IV, in order to speed up the compression process within the HyperLCA algorithm, the operations executed by the HyperLCA Transform have been parallelized using NVIDIA CUDA in such a way that they can be faster executed in the LPGPUs contained in the Jetson TK1 and Jetson TX2 developer boards. These operations have been coupled in seven different kernels, described in Section IV-B. The efficiency of these kernels directly affects the efficiency of the entire compression process for all the parallelized models carried out in this paper. Due to this reason, we have first verified the efficiency of these kernels before going into further evaluations for the entire compression process.

First of all, the developed kernels can be separated in two different groups. The first one, formed by the *cast to float kernel*, the *centroid calculation kernel*, and the *subtract centroid kernel*, includes those kernels that are executed just once per image block. The second group is formed by the kernels that are executed $p_{\max}$ times per image block, which are the *brightness vector calculation kernel*, the *maximum brightness calculation kernel*, the *projection vector calculation kernel*, and the *subtract information kernel*. The efficiency of the kernels included in the second group will have a more relevant impact in the efficiency of the entire compression process since they are launched more times.

In order to evaluate the efficiency of all these kernels, the NVIDIA profiling tool, included in the CUDA toolkit, has been used. This tool allows profiling the developed kernels extracting many different metrics. We have selected just three of these metrics in order to estimate if the SMs that conform the LPGPU are being efficiently used.

1) *Achieved occupancy (AO):* This metric measures the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor [36]. Its maximum value is 1. As described in Section III-A, an SM contains one or more warp schedulers. Each warp scheduler attempts to issue instructions from a warp on each clock cycle. In order to sufficiently hide latencies between dependent instructions, each scheduler must have at least one warp eligible to issue an instruction every clock cycle. Maintaining as many active warps as possible (a high occupancy) throughout the execution of the kernel helps to avoid situations where all warps are stalled and no instructions are issued. Hence, higher AO values will probably result in more efficient kernels.
2) *SM efficiency (SME):* This metric measures the percentage of time that at least one warp is active on a specific multiprocessor [36]. Its maximum value is 100%. A warp is active from the time it is scheduled on a multiprocessor until it completes the last instruction. Each warp scheduler maintains its own list of assigned active warps. This assignment of warps to the schedulers is done once at the

TABLE VI
PROFILING RESULTS OF THE KERNELS EXECUTED JUST ONCE PER IMAGE BLOCK

Cast to Float	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.85	0.85	0.85	0.83	0.82	0.81
SME (%)	96.15	92.64	86.53	96.25	92.32	85.19
WEE (%)	100.00	100.00	100.00	100.00	100.00	100.00
Centroid Calculation	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.99	0.99	0.98	0.92	0.96	0.96
SME (%)	99.58	99.36	99.12	98.59	98.22	98.08
WEE (%)	98.96	98.91	98.89	97.28	97.26	97.01
Subtract Centroid	Jetson TK-1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.83	0.83	0.83	0.83	0.81	0.81
SME (%)	96.47	93.28	87.67	96.52	92.74	86.07
WEE(%)	93.75	93.75	93.75	93.75	93.75	93.75

TABLE VII
PROFILING RESULTS OF THE KERNELS EXECUTED $p_{\max}$ TIMES PER IMAGE BLOCK

Brightness Vector Calc.	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.98	0.98	0.98	0.96	0.95	0.95
SME (%)	99.54	99.09	98.21	98.52	97.62	95.66
WEE %)	100.00	100.00	100.00	99.59	99.59	99.59
Max. Brightness Calc.	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.50	0.50	0.50	0.49	0.49	0.49
SME (%)	68.95	68.53	68.29	39.28	38.12	37.80
WEE (%)	98.55	98.52	98.51	98.70	98.67	98.66
Projection Vector Calc.	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.98	0.98	0.97	0.96	0.96	0.96
SME (%)	99.56	99.12	98.29	98.77	97.69	95.46
WEE (%)	100.00	100.00	100.00	99.64	99.64	99.64
Subtract Information	Jetson TK1			Jetson TX2		
BS	1024	512	256	1024	512	256
AO	0.99	0.98	0.98	0.95	0.94	0.95
SME (%)	99.65	99.34	98.74	99.15	98.17	97.26
WEE (%)	100.00	100.00	100.00	100.00	100.00	99.99

time a warp becomes active and it is valid for the lifetime of the warp. More active warps might allow hiding warp latencies more efficiently [37]. Hence, higher values for this metric will probably result in more efficient kernels.

3) *Warp execution efficiency (WEE):* This metric measures the ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor [36]. Its maximum value is 100%. CUDA devices operate most efficiently when all threads in a warp are enabled. There are two reasons that may disable some threads within a warp: being inactive, and being predicated off. The first one occurs when the thread BS defined for one specific kernel is not a multiple of the warp size. In this situation, the last warp of each block will have inactive threads. The second one occurs when there are divergent branches for a particular warp. In this situation, the separate paths taken by the threads of a particular warp must be serialized, and threads are disabled for paths they do not take. For each instruction executed, the ratio of threads enabled in the warp to full warp size is called control flow efficiency. The ideal control flow efficiency is 100% [38]. The WEE metric measures the average value of this ratio for all the executed instructions in the SM. Hence, its ideal value should be also 100%.

Table VI displays the values obtained by the profiling tool for these three metrics for the kernels that are executed just one time per image block. Similarly, Table VII shows the values obtained for the kernels executed $p_{\max}$ times per image block. These tables show the average results obtained for all the images in the described test bench. In order to measure the efficiency of the different kernels according to the different input parameters of the HyperLCA algorithm, the BS has been set to 1024, 512, and 256. The BS directly affects the efficiency of the kernels since it is extremely related to the fixed number of threads per CUDA thread block introduced for launching the different kernels, as Table III shows. On the contrary, the input CR and N_{bits} parameters have not been considered in these experiments. These two parameters only affect the calculation of the $p_{\max}$ value, as shown in (1). Accordingly, these parameters determine the number of times that the kernels are executed, but do not modify the kernels performance. Additionally, the N_{bits} parameter is also used for scaling the *V* vectors, as described in Section II-D1. However, this only affects the value used for multiplying the components of the *V* vectors, but the operations are exactly the same.

In general, the profiling results displayed in Tables VI and VII show very high values for the three metrics used for all the kernels, what suggests that the developed kernels are able to efficiently execute the operations of the HyperLCA Transform. The only kernel with relatively low AO and SME values is the *max brightness calculation* kernel. This is due to the fact that this kernel is only used for calculating the maximum value of a single vector of BS components. Hence, a maximum number of 1024, 512, and 256 threads are used for this kernel, according to the specified BS, what results in only 32, 16, and 8 warps being used, respectively. It can also be observed that bigger BS values tend to produce more efficient results. Basically, the best metric values are obtained when the image blocks correspond to entire hyperspectral frames (BS $= 1024$) and the worst results when independently processing quarters of the hyperspectral frames (BS $= 256$). Anyway, profiling results obtained using BS $= 1024$ are very close to the optimal ones, not very far from those issued using BS $= 512$. From all this, we can infer that bigger BS values would not bring much better results in terms of kernel efficiency.

D. Evaluation of the Maximum Frame Rate Obtained With the Different Parallelization Models

The experiments covered in this section evaluate the speedup achieved by the parallel implementations of the HyperLCA

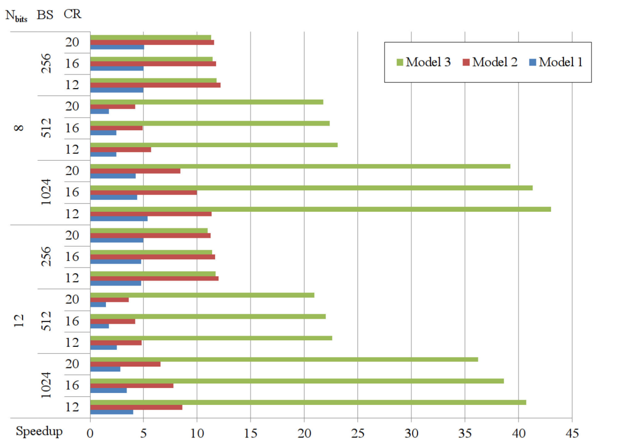


Fig. 8. Speed-up obtained with the described parallel models with respect to the reference version of the HyperLCA compressor in the Jetson TK1 NVIDIA board.

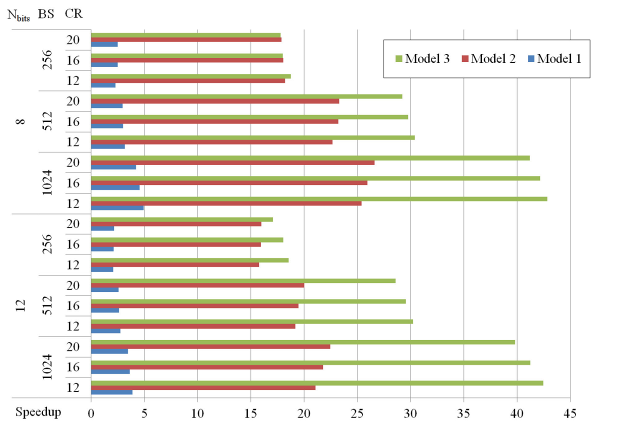


Fig. 9. Speed-up obtained with the described parallel models with respect to the reference version of the HyperLCA compressor in the Jetson TX2 NVIDIA board.

compressor developed in this paper in relation with the reference model. These implementations have been described in Section IV-D as *Parallel Model 1*, *Parallel Model 2*, and *Parallel Model 3*. These experiments also measure the maximum number of hyperspectral frames that these implementations are able to compress every second, according to the different configurations of the HyperLCA compressor. The maximum compression frame rate is a key factor for the utility of this research paper in the targeted application.

As done in Section V-C, the CR parameter has been set to 12, 16, and 20, the BS has been set to 1024, 512, and 256 and the N_{bits} parameter has been set to 12 and 8. This results in 18 different configurations of the HyperLCA compressor. The four images included in the previously described test bench have been compressed 10 times each one, using these 18 configurations for the three parallel implementations of the HyperLCA compressor. Figs. 8 and 9 graphically show the average speed-up obtained by the parallel implementations of the compressor in relation to its reference version, for each configuration of the HyperLCA algorithm. The speed-up was calculated as the ratio between the average time spent by each parallel model and the reference model to complete the entire compression process under the same parameter settings of the HyperLCA algorithm. The average time was calculated performing 40 simulations for each configuration, which means 10 simulations for each data set. Specifically, Fig. 8 shows the results obtained in the Jetson TK1 developer kit, whereas Fig. 9 shows the results obtained in the Jetson TX2.

Several conclusions can be drawn from these two figures. First of all, it can be observed that the *Parallel Model 1*, labeled as *Model 1*, is already 3 to 5 times faster than the reference version for both targeted boards. This speed-up is obtained just by executing the HyperLCA Transform in the corresponding LPGPU, using the developed kernels, as described in Fig. 4. Second, it is also appreciable that this speed-up considerably increases for the *Parallel Model 2* and *Parallel Model 3*, labeled as *Model 2* and *Model 3*, respectively, being *Model 3* generally faster than *Model 2* in both boards.

Further conclusions can be drawn by making a deeper analysis of these results. As explained in Section IV, *Model 1*, *Model 2*, and *Model 3* introduce different levels of parallelism, as described in Figs. 4, 5, and 6, respectively. These parallelism strategies reduce the time required for compressing the hyperspectral data. However, for being able to apply these parallel models and make use of the LPGPUs, the image blocks as well as the HyperLCA Transform outputs need to be copied to different parts of the memory, as described in Section IV-B. These copies are not needed in the reference model, and hence, negatively affect the obtained speed-up. Accordingly, the obtained speed-up corresponds to the time saved by applying the different parallelization strategies minus the time spent in making the copies of the image blocks and the HyperLCA Transform results. The time spent in each of these tasks depends on the specified configuration of the HyperLCA compressor and the characteristics of the Jetson TK1 and Jetson TX2 boards.

The impact of the N_{bits} parameter is not so high in the achieved speed-up. Basically, this parameter is used for calculating the $p_{\max}$ value, which determines the number of iterations of the HyperLCA transform, and hence, the number of times that the kernels are launched. $N_{\text{bits}} = 8$ produces higher $p_{\max}$ values than $N_{\text{bits}} = 12$. The time saved in each kernel execution should be the same with independence of the number of times that the kernel is executed, hence, the speed-up should also be the same. However, the memory copies of the image blocks and the HyperLCA Transform results are executed just once per image block. Due to this reason, the time lost in memory transfers is proportionally smaller in relation with the time required by the HyperLCA Transform for processing the data when more iterations are to be executed (higher $p_{\max}$ values). Accordingly, the achieved speed-up is slightly higher for $N_{\text{bits}} = 8$ than for $N_{\text{bits}} = 12$. Something similar happens with the CR input parameter. Higher CR values result in lower $p_{\max}$ values and less iterations in the HyperLCA transform for each image block.

Additionally, the CR and N_{bits} values also affect the amount of data produced as result in the HyperLCA transform and hence, higher CR and N_{bits} values slightly decrease the amount of data to be copy for each image block. Due to these reasons, the

TABLE VIII
EVALUATION OF THE MAXIMUM FRAME RATES OBTAINED USING THE JETSON TK1 AND TX2 NVIDIA BOARDS

			Jetson TK1								Jetson TX2							
Input parameters			Time (s)				Max Frame Rate				Time (s)				Max Frame Rate			
N_{bits}	BS	CR	Ref.	Mod. 1	Mod. 2	Mod. 3	Ref.	Mod. 1	Mod. 2	Mod. 3	Ref.	Mod. 1	Mod. 2	Mod. 3	Ref.	Mod. 1	Mod. 2	Mod. 3
12	1024	12	84.32	20.96	9.78	2.07	12	49	105	495	77.91	20.02	3.70	1.83	13	51	277	558
		16	65.66	19.07	8.43	1.70	16	54	122	603	60.66	16.72	2.78	1.47	17	61	368	697
		20	53.26	18.99	8.10	1.47	19	54	127	697	49.16	14.19	2.19	1.24	21	72	468	829
	512	12	71.96	28.86	15.01	3.18	14	35	72	322	65.41	23.88	3.41	2.16	16	43	300	473
		16	59.44	33.78	14.11	2.70	17	30	76	379	53.92	20.45	2.77	1.82	19	50	370	561
		20	46.93	31.45	12.92	2.24	22	33	87	457	42.50	16.56	2.12	1.49	24	62	483	689
	256	12	59.78	12.53	4.97	5.11	17	82	206	201	53.94	25.99	3.42	2.91	19	39	300	352
		16	47.09	9.86	4.03	4.14	22	104	254	248	42.39	20.13	2.66	2.35	24	51	385	436
		20	34.40	6.94	3.06	3.14	30	148	335	327	30.87	14.26	1.93	1.81	33	72	530	567
8	1024	12	114.05	21.25	10.04	2.65	9	48	102	386	105.21	21.35	4.14	2.46	10	48	247	417
		16	89.50	20.33	8.98	2.17	11	50	114	473	82.60	18.10	3.18	1.96	12	57	322	523
		20	71.10	16.65	8.43	1.81	14	62	122	565	65.63	15.55	2.46	1.59	16	66	416	643
	512	12	95.88	39.31	16.82	4.14	11	26	61	247	87.25	27.38	3.85	2.87	12	37	266	357
		16	71.17	28.93	14.51	3.18	14	35	72	322	64.78	21.58	2.79	2.18	16	48	367	471
		20	58.81	33.54	14.02	2.70	17	31	76	379	53.43	18.17	2.29	1.83	19	57	448	560
	256	12	71.65	14.41	5.89	6.05	14	71	174	169	64.82	28.53	3.56	3.45	16	36	288	296
		16	52.86	10.61	4.49	4.62	19	97	228	221	47.68	19.14	2.64	2.65	21	54	388	387
		20	46.60	9.24	4.03	4.12	22	111	254	248	41.99	16.93	2.35	2.36	24	61	436	434

variations in the speed-up obtained when using different CR and N_{bits} values will depend in how much time is saved in the HyperLCA Transform operations using these parallel models and how much time is lost in memory transfers. This also depends on the characteristics of the hardware. For instance, higher CR values produce faster results in the Jetson TX2 board when using *Model 2*, while it produces slower results in the Jetson TK1 using the same parallel model. This, together with the fact that the speed-up obtained by *Model 2* in the Jetson TX2 is much higher than the speed-up obtained with this parallel model in the Jetson TK1, suggests that the time spent in memory transfers has a higher impact in the Jetson TK1 than in the Jetson TX2 board.

Finally, when using *Model 3*, described in Fig. 6, the compression processes are pipelined as well as the memory transfers, and hence, the time spend in memory transfers can be neglected. This model produces the highest speed-ups, especially for bigger image blocks (BS = 1024). Additionally, the speed-up obtained by *Model 3* is much higher than the speed-up obtained by *Model 2* for bigger image blocks ($\text{BS} = 1024$), but it is almost the same for smaller blocks ($\text{BS} = 256$) in both boards. It is because when a kernel or a memory transfer are launched, the consumed time is not only spent on executing the kernel/transferring the data itself but also, on setting up and launching the instructions to do this. This means that the time consumed in transferring image blocks of 256 pixels is negligible in relation to the overhead of launching the memory transfers and the additional logic required in *Model 3*. However, the time required for transferring image blocks of 1024 pixels is considerably higher than the overhead of initializing the copy and the additional logic required logic in *Model 3*. On the basis that Model 2 and 3 only differ in the pipeline of the data transfers, we can conclude that bigger the BS, better the performance of *Model 3* in terms of speed-up than *Model 2* since it better hides the extra time required to transfer bigger blocks of pixels.

In addition to the speed-up results, graphically shown in Figs. 8 and 9, Table VIII shows the average results obtained for all the images and simulations executed in the Jetson TK1 and TX2 boards, for each configuration of the HyperLCA algorithm. Columns labeled as *max frame rate* in this table show the average number of hyperspectral F/S that the developed parallel models, as well as the reference version of the HyperLCA compressor, are theoretically able to compress using the Jetson TK1 and Jetson TX2 developer kits. These maximum frame rates were estimated by a rule of three since columns labeled as *Time* show the average number of seconds required to compress an entire image of 1024 frames. Therefore, the maximum frame rate can be approximated by dividing these times by 1024 and rounding up to the nearest entire whole number.

As described in Section V-A, the maximum compression frame rate is a key factor for the utility of this research paper in the targeted application. Ideally, the achievable frame rate with the developed implementations of the HyperLCA compressor should be higher than 330 F/S. This is due to the fact that the employed hyperspectral sensor is able to provide up to 330 F/S when collecting 224 spectral bands, and even more when collecting less bands as we are doing. By achieving this frame rate, it can be guarantied that the collected hyperspectral data can be always compressed in real time, and the capturing frame rate can be specified according to the characteristics of the application (such as flight height, intensity of the light, area to cover, etc.) without being limited by the data compression process. It must be mentioned that the desirable frame represents the most extreme case since the employed frame rates are normally smaller. For instance, the images included in the test bench were collected at 150 and 200 F/S.

According to the results shown in Table VIII, it can be concluded that the parallel implementation of the HyperLCA compressor named *Model 3* executed in the Jetson TX2 board is able to achieve more than 330 F/S with independence of the configuration used. It is also observable that using bigger N_{bits} values, $N_{\mathrm{bits}} = 12$, always guarantee faster compression results (higher FPS). This is due to the fact that less V vectors are extracted by the HyperLCA Transform, what results in less iterations required by the HyperLCA Transform and less output results to be copied to the host. Additionally, there are less vectors to be processed by the entropy coder. *Model 2* executed in the Jetson TX2 board is also able to achieve more than 330 F/S for some configurations, specially for higher CRs (CR = 20 and CR = 16). The fact of obtaining faster results for higher CRs represents an extra advantage, since increasing the CR as the acquisition data rate increases the compressed data transfer and/or storage. It is also worth mentioning that compression frame rates higher than 330 F/S are only achievable using *Model 3* when using the Jetson TK1 developer kit. As it occurs with the Jetson TX2 board, the fastest results are obtained using $N_{\mathrm{bits}} = 12$, higher CRs, and bigger image blocks. Despite the achieved frame rates, using *Model 2* and *Model 3* do not surpass the desired 330 FPS for all the situations, especially in the Jetson TK1 board, the achieved FPS are still relatively high in relation with the frame rates typically used in the targeted applications.

The results shown in Table VIII also manifest that the achievable compression rate using the reference version of the HyperLCA compressor, serially executed in the CPU, is very poor in relation to the requirements of the targeted application. This fact clearly justifies the necessity of the development of parallel implementations of the algorithm in order to achieve real-time compression for being able to efficiently store and/or transfer the acquired hyperspectral data, as it has been done in this paper.

E. Evaluation of the HyperLCA Compression Performance for the Targeted Application

The goodness of the HyperLCA compressor for the application targeted in this paper has been also evaluated. For doing so, the test bench previously described has been compressed and decompressed using those configurations of the HyperLCA compression, which bring frame rates more close to the desired ones in both Jetson boards, that is, BS set to 1024 and 512, N_{bits} set to 12, and CR set to 16 and 20.

The compression performance of the HyperLCA algorithm has been evaluated by measuring the CR achieved as well as the information lost in the compression–decompression process. The achieved CR has been measured in two different ways, calculating the relation between original data volume and the compressed data volume, CR, and measuring the number of bits per pixel per band (bpppb) used for representing the image, in average, according to the compressed image size. Higher CR and lower bpppb values indicate that a higher compression has been achieved.

The information lost in the compression–decompression process has been measured using four different quality metrics: the *SNR*, the root mean squared error (RMSE), the peak SNR (PSNR), and the MAD, as it was done in [12]. The MAD evaluates the amount of information lost for the worst represented value of the entire image. In order to know if the obtained MAD value indicates a good compression performance or not, it is important to consider the maximum MAD value that could

TABLE IX
EVALUATION OF THE QUALITY OF THE HYPERLCA COMPRESSION RESULTS

Input parameters			Outputs					
Nbits	BS	CR	CR	bpppb	SNR	MAD	RMSE	PSNR
12	1024	16	23.79	0.50	41.97	32.50	3.52	61.32
		20	31.05	0.39	41.06	41.25	3.91	60.41
	512	16	23.56	0.51	42.14	30.50	3.45	61.49
		20	31.71	0.38	41.24	40.00	3.83	60.59

be obtained for the targeted image. The sensor described in Section V-A, which collected the images used in these experiments, provides the hyperspectral data using 16 bits per pixel per band. However, just 12 of these 16 bits are actually used for representing the image values. This means that the dynamic range of the sensor is $2^{12} = 4096$. Accordingly, the maximum possible (and worst) MAD value is 4095. The RMSE evaluates the average information lost in the entire image. As well as for the MAD metric, the maximum possible (and worst) RMSE value for these images is 4095. The SNR and PSNR metrics also evaluate the average information lost in the entire image, however, these metrics already compare the average error with the maximum possible error, making different considerations, in such a way that higher SNR and PSNR values indicate better compression performance. The SNR and PSNR metrics are calculated in decibels.

Table IX shows the average results obtained for each configuration of the HyperLCA compressor using the four hyperspectral images shown in Fig. 7. Different conclusions can be drawn from these results. First of all, the CR achieved by the compressor, Output CR, is always higher than the specified Input CR. This is an important advantage considering that the Input CR is interpreted as the minimal desired CR. Second, it is also demonstrated that the HyperLCA compressor is able to provide very high CRs, since the compressed images are at least 23 times smaller than the original images for all the tested configurations (less than 0.5 bpppb). Finally, and more important, these CRs have been achieved losing very few information, as shown by the values obtained for the different quality metrics.

VI. CONCLUSION

In this paper, we present the implementation of the HyperLCA lossy compressor onto the NVIDIA Jetson TK1 and Jetson TX2 boards, which embed LPGPUs. This paper responses to the necessity of being able to compress hyperspectral images in real time for remote sensing applications with limitations on computational resources due to power, weight, and/or space limitations. In particular, this paper focuses on processing hyperspectral imaging data collected by a drone at very high frame rates for smart farming applications.

The HyperLCA algorithm has been selected due to the several advantages that it offers. Some advantages are related to its compression performance since this algorithm is able to achieve very high CRs at reasonable high rate distortion relations and preserving the most relevant information for the ulterior hyperspectral imaging applications. Other advantages are related to its low computational complexity and high level of parallelism, what allows making an efficient use of the computational resources of the Jetson TK1 and TX2 boards for speeding up the compression process. However, its more interesting advantage is its capability to independently compress blocks of hyperspectral pixels without any required spatial alignment between them, which makes it most suitable for compressing hyperspectral frames collected by pushbroom/whiskbroom sensors as soon as they are captured.

In this paper, we have implemented the HyperLCA compressor taking into account the limitations on the computational resources, the high frame rates imposed by the targeted application and that the compression process is just one of the many tasks that are running at the same in the Jetson boards. For these reasons, we have first analyzed the different stages of the HyperLCA compressor in terms of FLOPs and execution times in order to identify those stages more suitable for being accelerated in the embedded LPGPUs. Second, we have studied the key concepts that need to be considered for being able to develop efficient kernels and make the best use of the LPGPU resources. Third, we have exhaustively described there different implementation models of the whole compression process using additional parallelization strategies to the already considered in the CUDA programming model. These strategies are focused on exploiting the parallelism of the HyperLCA compressor beyond the thread level parallelism inherent to the GPU programming model. Concretely, pipelining the execution of the compression stages of the HyperLCA algorithm for the different blocks of pixels as well as the communications and memory transfers.

Additionally, multiple experiments have been carried out using real hyperspectral images. The obtained results have been exhaustively evaluated and compared providing crucial information for the potential designers of the next generation of hyperspectral imaging systems. Particularly, the experiments carried out in this paper are oriented to the necessities imposed by a specific smart farming application although all drawn conclusions are extrapolated to other fields in which remotely sensed hyperspectral images are to be compressed in real time. Specifically, the obtained results verify that the developed implementations are able to carry out the real-time compression of the hyperspectral data provided by the acquisition system used in the targeted application, reaching compression rates bigger than 330 F/S.

REFERENCES

[1] P. S. Thenkabail, J. G. Lyon, and A. Huete, *Fundamentals, Sensor Systems, Spectral Libraries, and Data Mining for Vegetation*. Boca Raton, FL, USA: CRC Press, 2018.

[2] T. Adão *et al.*, "Hyperspectral imaging: A review on UAV-based sensors, data processing and applications for agriculture and forestry," *Remote Sens.*, vol. 9, no. 11, 2017, Art. no. 1110.

[3] T. Lillesand, R. W. Kiefer, and J. Chipman, *Remote Sensing and Image Interpretation*. Hoboken, NJ USA: Wiley, 2014.

[4] C.-I. Chang, *Hyperspectral Data Exploitation: Theory and Applications*. Hoboken, NJ, USA: Wiley, 2007.

[5] G. Motta, F. Rizzo, and J. A. Storer, *Hyperspectral Data Compression*. New York, NY, USA: Springer, 2006.

[6] B. Penna, T. Tillo, E. Magli, and G. Olmo, "Transform coding techniques for lossy hyperspectral data compression," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 5, pp. 1408–1421, May 2007.

[7] B. Penna, T. Tillo, E. Magli, and G. Olmo, "Progressive 3-D coding of hyperspectral images based on JPEG 2000," *IEEE Geosci. Remote Sens. Lett.*, vol. 3, no. 1, pp. 125–129, Jan. 2006.

[8] M. W. Marcellin and D. S. Taubman, *JPEG2000: Image Compression Fundamentals, Standards, and Practice Volume 642 of Kluwer International Series in Engineering and Computer Science*. New York, NY, USA: Springer, 2002.

[9] L. Chang, C.-M. Cheng, and T.-C. Chen, "An efficient adaptive KLT for multispectral image compression," in *Proc. 4th IEEE Southwest Symp. Image Anal. Interpretation*, 2000, pp. 252–255.

[10] P. Hao and Q. Shi, "Reversible integer KLT for progressive-to-lossless compression of multiple component images," in *Proc. Int. Conf. Image Process.*, 2003, vol. 1, pp. I–633.

[11] A. Abrardo, M. Barni, and E. Magli, "Low-complexity predictive lossy compression of hyperspectral and ultraspectral images," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, IEEE, 2011, pp. 797–800.

[12] R. Guerra, Y. Barrios, M. Díaz, L. Santos, S. López, and R. Sarmiento, "A new algorithm for the on-board compression of hyperspectral images," *Remote Sens.*, vol. 10, no. 3, 2018, Art. no. 428.

[13] NVIDIA Jetson TK1 Information. [Online]. Available: https://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html (Accessed on: Mar. 2019).

[14] NVIDIA Jetson TX2 Information. [Online]. Available: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-tx2/ (Accessed on: May 2019).

[15] J. M. Nascimento, M. Véstias, and R. Duarte, "Hyperspectral compressive sensing: a low-power consumption approach," in *Proc. High-Perform. Comput. Geosci. Remote Sens. VIII*, 2018, vol. 10792.

[16] J. M. Nascimento and G. Martin, "On the use of jetson TX1 board for parallel hyperspectral compressive sensing," in *Proc. High-Perform. Comput. Geosci. Remote Sens. VII*, 2017, vol. 10430.

[17] J. M. Nascimento and G. Martin, "Hyperspectral compressive sensing on low energy consumption board," in *Proc. IEEE Int. Geosci. Remote Sens. Symp.*, 2018, pp. 5065–5068.

[18] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.

[19] Consultative Committee for Space Data Systems (CCSDS), "*Lossless Multispectral & Hyperspectral Image Compression, Issue 1, Recommendation for Space Data System Standars (Blue Book), CCSDS 123.0-B-1*," Sep. 2017. [Online]. Available: https://public.ccsds.org/Pubs/122x1b1.pdf (Accessed on: Mar. 2019).

[20] M. Ciznicki, K. Kurowski, and A. J. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," *J. Appl. Remote Sens.*, vol. 6, no. 1, 2012, Art. no. 061507.

[21] L. Santos, E. Magli, R. Vitulli, J. F. López, and R. Sarmiento, "Highly-parallel GPU architecture for lossy hyperspectral image compression," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 6, no. 2, pp. 670–681, Apr. 2013.

[22] B. Hopson, K. Benkrid, D. Keymeulen, and N. Aranki, "Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU & multicore processor systems," in *Proc. NASA/ESA Conf. Adaptive Hardware Syst.*, 2012, pp. 107–114.

[23] D. Keymeulen, N. Aranki, B. Hopson, A. Kiely, M. Klimesh, and K. Benkrid, "GPU lossless hyperspectral data compression system for space applications," in *Proc. IEEE Aerosp. Conf.*, 2012, pp. 1–9.

[24] R. Davidson and C. Bridges, "Adaptive multispectral GPU accelerated architecture for earth observation satellites," in *Proc. IEEE Int. Conf. Imag. Syst. Techn.*, 2016, pp. 117–122.

[25] R. Davidson and C. Bridges, "GPU accelerated multispectral EO imagery optimised CCSDS-123 lossless compression implementation," in *Proc. IEEE Aerosp. Conf.*, 2017, pp. 1–12.

[26] R. Guerra, L. Santos, S. López, and R. Sarmiento, "A new fast algorithm for linearly unmixing hyperspectral images," *IEEE Trans. Geosci. Remote Sens.*, vol. 53, no. 12, pp. 6752–6765, Dec. 2015.

[27] R. Guerra, S. López, and R. Sarmiento, "A computationally efficient algorithm for fusing multispectral and hyperspectral images," *IEEE Trans. Geosci. Remote Sens.*, vol. 54, no. 10, pp. 5712–5728, Oct. 2016.

[28] M. Díaz, R. Guerra, S. López, and R. Sarmiento, "An algorithm for an accurate detection of anomalies in hyperspectral images with a low computational complexity," *IEEE Trans. Geosci. Remote Sens.*, vol. 56, no. 2, pp. 1159–1176, Feb. 2018.

[29] Consultative Committee for Space Data Systems (CCSDS), "*Blue Books: Recommended Standards.*" [Online]. Available: https://public.ccsds.org/Publications/BlueBooks.aspx (Accessed on: Mar. 2019).

[30] P. G. Howard and J. S. Vitter, "Fast and efficient lossless image compression," in *Proc. Data Compression Conf.*, 1993, pp. 351–360.
[31] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. Hoboken, NJ USA: Wiley, 2014.
[32] NVIDIA Developer. CUDA Toolkit Documentation. Features and Technical Specifications - Table 14. Technical Specifications per Compute Capability. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications (Accessed on: Mar. 2019).
[33] Specim FX1 Series hyperspectral cameras. [Online]. Available: http://www.specim.fi/fx/ (Accessed on: March 2019).
[34] NVIDIA Jetson TK1 Information. [Online]. Available: https://elinux.org/Jetson_TK1 (Accessed on: Mar. 2019).
[35] NVIDIA Developer. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/ (Accessed on: Mar. 2019).
[36] NVIDIA Corporation. Profiler User′s Guide. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference (Accessed on: Mar. 2019).
[37] NVIDIA Corporation. Developer tools. Issue Efficiency. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm (Accessed on: Mar. 2019).
[38] NVIDIA Corporation. Developer tools. Instruction Count. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/instructioncount.htm (Accessed on: Mar. 2019).

María Díaz was born in Spain in 1990. She received the Industrial Engineer degree from the University of Las Palmas de Gran Canaria, Las Palmas, Spain, in 2014. In 2017, she received the master's degree in system and control engineering imparted jointly by the Complutense University of Madrid, Madrid, Spain, and the National University of Distance Education, Madrid. She is currently working toward the Ph.D. degree with the University of Las Palmas de Gran Canaria, developing her research activities at the Integrated Systems Design Division, Institute for Applied Microelectronics.

Her research interests include image and video processing, development of highly parallelized algorithms for hyperspectral images processing, and hardware implementation.

Raúl Guerra was born in Las Palmas de Gran Canaria, Spain, in 1988. He received the Industrial Engineering degree from the University of Las Palmas de Gran Canaria, Las Palmas, Spain, in 2012, the master's degree in telecommunications technologies from the Institute of Applied Microelectronics, Las Palmas, and the Ph.D. degree in telecommunications technologies from the University of Las Palmas de Gran Canarias in 2017, funded by the Institute of Applied Microelectronics.

His current research interests include the parallelization of algorithms for multispectral and hyperspectral images processing and hardware implementation.

Pablo Horstrand was born in Las Palmas de Gran Canaria, Las Palmas, Spain, in 1986. He received the double degree in industrial engineering and electronics and control engineering, in 2010, and the master's degree in telecommunication technologies in 2011, both from the University of Las Palmas de Gran Canaria, where he is currently working toward the Ph.D. degree in telecomunication technologies.

He worked between 2011 and 2017 with ABB Switzerland, first in the Minerals and Printing, Drives Department, and last in the Traction Department, Aargau, Switzerland. During 2018, he was with the Royal Military Academy, Brussels, Belgium, as part of his Ph.D., doing research in the Signal and Image Center Department.

Ernestina Martel was born in Las Palmas de Gran Canaria, Spain, in 1968. She received the Ph.D. degree in computer science from the University of Las Palmas de Gran Canaria, Las Palmas, Spain, in 2004.

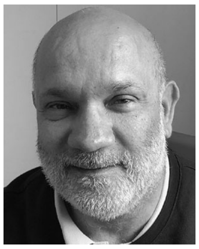
She currently works as an Associate Professor with the University of Las Palmas de Gran Canaria. She has been a member of the Institute for Applied Microelectronics (IUMA) since 1995. She collaborated with the Laboratory of Distributed Systems, Technical University of Madrid, Madrid, Spain, for five years (2004–2009) in different research projects funded by the European Community and Spanish Government. She is currently developing her research activities at the Integrated Systems Design Division, IUMA. Her research interests include the area of architecture, implementation, performance evaluation, and algorithms for parallel computing.

Dr. Martel received a Special Award from ANECA (Spanish agency for the evaluation of the research activity) for her research on the semantic recommendation systems in 2009.

Sebastián López (M'08–SM'15) was born in Las Palmas de Gran Canaria, Spain, in 1978. He received the Degree in electronic engineering from the University of La Laguna, San Cristobal de La Laguna, Spain, in 2001, and the Ph.D. degree in electronic engineering from the University of Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain, in 2006.

He is currently an Associate Professor with the University of Las Palmas de Gran Canaria, where he is involved in research activities with the Integrated Systems Design Division, Institute for Applied Microelectronics. He has coauthored more than 120 papers in international journals and conferences. His research interests include real-time hyperspectral imaging, reconfigurable architectures, high-performance computing systems, and image and video processing and compression.

Dr. López was a recipient of regional and national awards during his electronic engineering degree. He is also an Associate Editor for the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING, *MDPI Remote Sensing, and the Mathematical Problems in Engineering* Journal. He was an Associate Editor for the IEEE TRANSACTIONS ON CONSUMER ELECTRONICS from 2008 to 2013. He also serves as an active reviewer for different JCR journals and as a program committee member of a variety of reputed international conferences. Furthermore, he was one of the Program Chairs of the IEEE Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing in its 2014 edition and of the SPIE Conference of High Performance Computing in Remote Sensing from 2015 to 2018. Moreover, he has been the Guest Editor of different special issues in JCR journals related with his research interests.



José F. López received the M.S. degree in physics (specializing in electronics) from the University of Seville, Seville, Spain, and the Ph.D. degree from the University of Las Palmas de Gran Canaria (ULPGC), Las Palmas, Spain.

He has conducted his investigations at the Institute for Applied Microelectronics, where he acts as the Deputy Director since 2009. He currently lectures with the School of Telecommunication and Electronics Engineering and the M.Sc. Program of IUMA, in the ULPGC. He was with Thomson Composants Microondes, Orsay, France, in 1992. In 1995, he was with the Center for Broadband Telecommunications, Technical University of Denmark, Lyngby, Denmark, and in 1996, 1997, 1999, and 2000, he was a Visiting Researcher with the Edith Cowan University, Perth, Western Australia. His main areas of interest include the field of image processing, UAVs, hyperspectral technology, and their applications.

Dr. Lopez has been actively enrolled in more than 25 research projects funded by the European Community, Spanish Government, and international private industries. He has written around 90 papers in national and international journals and conferences. He was awarded by the University of Las Palmas de Gran Canaria for his research in the field of high speed integrated circuits.

Roberto Sarmiento received the Ph.D. degree in electronic engineering from the University of Las Palmas de Gran Canaria (ULPGC), Las Palmas, Spain, in 1991.

He is a Full Professor with the Telecommunication Engineering Faculty, University of Las Palmas de Gran Canaria, in the area of electronic engineering. He contributed to the birth of this center in 1989. He was the Dean of the faculty from 1994 to 1998 and Vice-Chancellor for Academic Affairs and Staff with the ULPGC from 1998 to 2003. In 1993, he was a Visiting Professor with the University of Adelaide, Adelaide, South Australia, and later at the Edith Cowan University, Perth, Western Australia. He cofounded the Institute for Applied Microelectronics and was the Director of the Integrated Systems Design Division of this institute. Since 1990, he has published more than 40 journal papers and book chapters and more than 120 conference papers. His research interests include multimedia processing and video coding standard systems, reconfigurable architectures and real-time processing, and compression of hyperspectral imaging.

Dr. Sarmiento has been awarded with four six years research periods by the National Agency for the Research Activity Evaluation in Spain. He has participated in more than 35 projects and research programs funded by public and private organizations, among which he has been a Leader Researcher in 16 of them. He has conducted several agreements with companies for the design of high-performance integrated circuits, being the most remarkable the collaboration with Vitesse Semiconductor Corporation, California, and Thales Alenia Space, Spain.